User-based Secure File Access and Sharing under Constrained Environment

Gaurav Kuwar

February 24, 2025

1 Abstract

This paper dives into a project I started during HackNYU—a secure file access and sharing system for offline environments like family hard drives. What began as a simple idea to give users access to files on a local drive grew into a more complex challenge with added constraints: no internet, all cryptographic data staying on the drive, and a frontend that's lightweight and doesn't store any unique crypto data. Users can log in from any computer with just their username and password, and they can share files with others securely. To make this work, I built a cryptographic scheme using key wrapping and Schnorr signatures to manage file access and verify identities, plus a tamper-proof audit trail to catch any funny business. But there's a catch—the system is still vulnerable to a full replacement attack, which means we'll need to look into hardware solutions down the line. This project lays the groundwork for secure, offline file sharing, perfect for families or anyone needing local storage with user-based access.

2 Problem Description

The system is designed to provide secure, user-based file access and sharing on a local hard drive, operating entirely without internet connectivity. The core problem revolves around managing file access for multiple users in a constrained environment, while ensuring a bad actor cannot tamper with cryptographic data on the drive, which would allow them to gain unauthorized access to a user's files.

2.1 Key Requirements

The system must address the following requirements:

- **User-Account Based Access:** Each user has a unique account identified by a username and password. Users can log in from any computer with their credentials, and the system must support multiple users with distinct access permissions.
- **File Management:** Users can add and delete files. Additionally, users can grant or revoke access to their files for other users, ensuring secure file sharing.
- **Constrained Environment:** The system operates entirely offline, with no internet connectivity. All cryptographic data, including keys and access control lists, must reside on the hard drive. The frontend application is lightweight and contains no unique cryptographic data, allowing users to access the drive from any instance of the application.

2.2 Security Constraints

The system must operate under the following security constraints:

- **Single-User Access:** Only one user can access the drive and their files at any given time, and in between user sessions, we must assume that a bad actor has access to the encrypted files and cryptographic data on the hard drive. Hence, the user must be able to verify the authenticity of the changes made to the system.

- **Exposure to Bad Actors:** Between the logout of one user and the login of another, we must assume that a bad actor has access to the files and cryptographic data on the hard drive. The system must protect against unauthorized access, tampering, and impersonation during these transitions.

3 Proposed Solution

The system implements a cryptographic scheme to provide secure file access and sharing in a constrained, offline environment. The solution addresses key vulnerabilities such as unauthorized access, tampering, and impersonation through a combination of key wrapping, Schnorr signatures, and a tamper-proof audit trail.

3.1 Cryptographic Scheme

The system employs the following cryptographic primitives and operations, including a Diffie-Hellman key exchange scheme that enables secure file sharing between users by establishing shared secret keys for encrypting file access keys:

- u_i: Username of user i.

- p_i: Password of user i.

- p: A large prime number (2048 bits) for modular arithmetic.

- g: A primitive root of p, used as the base for key generation.

- $x_i = \text{HKDF}(u_i + p_i, \text{fixed\_salt})$: A user-specific private key derived from the username and password using a key derivation function (HKDF).

- $y_i = g^{x_i} \mod p$: A user-specific public key, used for key sharing and verification.

3.2 Key Management

Each file F_k is encrypted with a unique master key MK_k. The key management scheme ensures secure file access and sharing:

- **File Encryption:** Files are encrypted symmetrically using AES with a unique master key MK_k. This ensures that even if one file is compromised, others remain secure.

- **Key Wrapping:** When a file is created or added by user u_i, its master key MK_k is encrypted with x_i (the user's private key). This ensures that only u_i can access the file initially.

- **Key Sharing:** To share file F_k with user u_j, u_i computes a shared key $y_{ij} = y_j^{x_i} \mod p$ and encrypts MK_k with y_{ij}. User u_j can then decrypt the wrapped key by computing $y_{ij} = y_i^{x_j} \mod p$. This ensures secure key sharing without exposing MK_k directly.

- **Access Control List (ACL):** The ACL structure is defined as:

$$\text{acl} = \{u_i : \{u_j : E(\{F_0, \ldots F_k\}, y_{ij})\} \ldots\}$$

  where E is a symmetric encryption function. This structure ensures that only authorized users can access specific files.

3.3 Integrity Verification with Schnorr Signatures

To prevent impersonation and ensure the integrity of public keys y_i, the system uses Schnorr signatures:

- q: A prime divisor of $p - 1$.
- r_i: A random nonce generated using a secure random number generator.
- $R_i = g^{r_i} \mod p$: A commitment value used in the signature.
- $e_i = \text{SHA-256}(R_i, y_i) \mod q$: A challenge value derived from the commitment and public key.
- $s_i = (r_i + x_i \times e_i) \mod q$: The signature component.
- $\text{sig}(y_i) = (e_i, s_i)$: The Schnorr signature for y_i.

3.3.1 Verification Process

User u_j verifies the integrity of y_i as follows:

$$R'_i = g^{s_i} \times y_i^{-e_i} \mod p$$

$$e'_i = \text{SHA-256}(R'_i, y_i) \mod q$$

The signature is valid if $e'_i == e_i$. This ensures that y_i is bound to x_i, this signature was created by the user with private key x_i and this public key is not replaced by bad actor with their own.

3.4 Tamper-proof Audit Trail

To detect and prevent tampering, the system maintains a tamper-proof audit trail using hash chaining:

- **Initialization:** Start with $Y = \{k\}$, where k is a random number, and $L = \{H(Y)\}$.
- **Adding a User:** When a new user is added:

$$l_i = H(Y), \text{change log}, l_{i-1}$$

$$L.\text{append}((l_i, H(l_i)))$$

- **Verification:** To verify the integrity of the audit trail, users can recompute the hash chain starting from the most recent entry and compare it with the stored values. Additionally, a user can store an encrypted checkpoint with their private key, reducing the amount of computation required to verify the integrity of the audit trail at every session.

4 Limitations and Next Steps

4.1 Software Limitations

The current implementation faces a full replacement attack vulnerability, where a malicious actor could potentially replace the entire system, and a new user would have no way to verify the authenticity of the public keys. Additionally, I will be exploring potential software optimizations such as using a Merkle tree for the audit trail to reduce the amount of data that needs to be stored and verified.

4.2 Hardware Considerations

To address these limitations, we will be exploring hardware-based solutions.